RESEARCH-ARTICLE

# gShare: Efficient GPU Sharing with Aggressive Scheduling in Multi-tenant FaaS platform

**YANAN YANG**, China Telecom Corporation Limited, Beijing, Beijing, China

**ZHENGXIONG JIANG**, China Telecom Corporation Limited, Beijing, Beijing, China

**MEIQI ZHU**, China Telecom Corporation Limited, Beijing, Beijing, China

**HONGQIANG XU**, China Telecom Corporation Limited, Beijing, Beijing, China

**YUJUN WANG**, China Telecom Corporation Limited, Beijing, Beijing, China

**LIANG LI**, China Telecom Corporation Limited, Beijing, Beijing, China

**View all**

**Open Access Support** provided by:

**China Telecom Corporation Limited**

**Temple University**



**Published:** 22 March 2026

**Citation in BibTeX format**

ASPLOS '26: 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems
*March 22 - 26, 2026*
*PA, Pittsburgh, USA*

**Conference Sponsors:**
SIGOPS
SIGPLAN
SIGARCH

# gShare: Efficient GPU Sharing with Aggressive Scheduling in Multi-tenant FaaS platform

Yanan Yang
yangyn11@chinatelecom.cn
China Telecom Cloud Computing Research Institute
Beijing, China

Zhengxiong Jiang
jiangzx7@chinatelecom.cn
China Telecom Cloud Technology Co. Ltd.
Chengdu, China

Meiqi Zhu
zhumq@chinatelecom.cn
China Telecom Cloud Technology Co. Ltd.
Guangzhou, China

Hongqiang Xu
xuhq7@chinatelecom.cn
China Telecom Cloud Technology Co. Ltd.
Chengdu, China

Yujun Wang
wangyj3@chinatelecom.cn
China Telecom Cloud Technology Co. Ltd.
Guangzhou, China

Liang Li*
lil225@chinatelecom.cn
China Telecom Cloud Computing Research Institute
Beijing, China

Jiansong Zhang
zhangjs15@chinatelecom.cn
China Telecom Cloud Computing Research Institute
Beijing, China

Jie Wu
jiewu@temple.edu
China Telecom Cloud Computing Research Institute
Beijing, China
Temple University
Philadelphia, United States

## Abstract

Serving ML models with serverless computing has become increasingly popular in recent years. Many of today's cloud vendors have provided GPU functions to meet the performance requirements of different ML scenarios. However, existing production FaaS platforms suffer from GPU underutilization and high cloud costs due to poor GPU resource management. In this paper, we propose gShare, an on-demand and efficient GPU function management policy in FaaS platforms. gShare provides a fine-grained GPU virtualization solution for a VM-based multi-tenant FaaS environment. It further decouples the GPU resource from the existing CPU-oriented function management paradigm, enabling flexible GPU sharing across tenants. With a user-transparent vGPU remapping design and aggressive request scheduling policy, gShare can significantly improve the cost-efficiency of GPU functions without causing appreciable function performance degradation. Experimental results show that gShare can reduce GPU usage by 43%–63% compared to the baseline while meeting more than 95% of user latency targets. Compared with the state-of-the-art method, it can also reduce cloud costs by 24%-58% while maintaining better function performance, benefiting both the cloud provider and users.

***CCS Concepts:*** • **Computer systems organization** → **Cloud computing**.

***Keywords:*** FaaS-as-a-Service, GPU function, Hardware virtualization

**ACM Reference Format:**
Yanan Yang, Zhengxiong Jiang, Meiqi Zhu, Hongqiang Xu, Yujun Wang, Liang Li, Jiansong Zhang, and Jie Wu. 2026. gShare: Efficient GPU Sharing with Aggressive Scheduling in Multi-tenant FaaS platform . In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3779212.3790168

## 1 Introduction

Serverless computing, also known as Function-as-a-Service (FaaS), has emerged as a popular cloud paradigm in the past few years. Due to its ease of use and pay-as-you-go benefits, there has been a wide variety of workloads built in FaaS platforms, including video processing [25], data analytics [10, 39, 40, 49, 57, 68, 102], and web services [8, 90, 91, 108]. With the growing adoption of machine learning (ML) applications, serving ML models (a.k.a., inference) with FaaS functions is becoming increasingly popular recently [5, 34, 48, 71, 84, 103]. Major FaaS providers like AWS Lambda, Microsoft Azure

*Corresponding author: Liang Li



*ASPLOS '26, Pittsburgh, PA, USA*

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790168

Functions, and Alibaba Cloud Functions [16] all report that serverless model serving has been a common use case.

As model serving is always performed in real time with intensive computation and stringent latency Service-Level-objective (SLO). Many commercial FaaS platforms have offered GPU support to meet the growing demand for building fast, low-latency model-serving services. For instance, Azure Container Apps provides NVIDIA A100 GPUs for serverless model training and inference in some regions [9], and Alibaba Cloud Function supports configuring NVIDIA V100 GPUs for functions in units of 1GB of device memory [16]. In addition, according to a recent case study [95], serving models on GPUs often outperforms traditional cloud-based inference in terms of cost and scalability.

Unfortunately, we find the existing GPU function management in FaaS platforms performs inefficiently, making them suffer from high operating costs [36]. On the one hand, coarse-grained GPU provisioning in current FaaS platforms does not cater to the needs of many small models, leading to significant resource waste and GPU under-utilization. Although techniques like MPS [61] allow GPU sharing among functions, these solutions require kernels from different tenants run in the same context, which can not be used in a public cloud, where user functions usually run in individual VMs for better isolation and security. Other GPU partition approaches, such as MIG [60] and vGPU [64], either lack flexibility or face generality issues, making them unable to support on-demand GPU provisioning in a VM-based multi-tenant FaaS cluster. Even combined with single-instance multi-concurrency techniques [37, 81], inflexible GPU allocation also leads to high costs during workload bursts.

On the other hand, unlike CPUs that can be recycled in idle function instances, GPUs are coupled with function memory management and cannot be flexibly offloaded for reclamation, exacerbating resource inefficiency. Function prewarming and keep-live methods are widely used in FaaS platforms to help mitigate coldstart overhead [27, 75, 77, 98]. Specifically, FaaS providers use provisioned instances to speed up function startup. After each invocation, an idle function instance is cached for a period of time (e.g., 15 minutes in AWS Lambda [43]), thereby reducing coldstart invocations by reusing instances. As GPUs are much more expensive than CPUs and always have longer coldstart latency, prewarming or caching many GPU functions can lead to 10× higher cloud costs, according to our measurements.

Prior work aims to improve the revenues of FaaS platforms through preemptible GPU functions [16, 103]. Once enabled, users allow the FaaS platform to recycle GPUs from their keep-alive functions and receive a discount. Model swapping is a core technique. By keeping model data in host memory and dynamically swapping it onto the GPU when a request arrives, different functions can share a node's GPUs. Such a design increases GPU utilization but still falls short in high efficiency: Firstly, it relies on a GPU proxy-based control plane [23] to manage model data (e.g., loading& swapping), serves requests by interacting with the GPU clients inside functions, which incurs system overhead (7%-61% of performance fluctuations with 40% more forwarding delay). Secondly, model swapping may increase request latency by 10s to 100s of milliseconds; the simple heuristic scheduling in the existing design is oblivious to function behavior, resulting in either low GPU utilization or SLO violations. Moreover, resource waste from coarse-grained GPU provisioning still exists.

In this paper, we argue for a domain-specific GPU function management policy for multi-tenant FaaS clusters with several insights: Firstly, GPU resources should be provided in a fine-grained manner, as with CPU functions (e.g., a minimum of 0.05 vCPU), thereby precisely matching users' requirements and minimizing resource waste. Second, GPUs should be decoupled from the existing function resource management scheme to support flexible recycling and sharing, which can bring an opportunity for FaaS providers to reduce resource cost from resource sharing. Moreover, some serving functions may have smaller model sizes and more lenient latency tolerances (e.g., batch requests or asynchronous invocations), which can be exploited to explore more aggressive GPU sharing policies, thereby improving cost efficiency in FaaS platforms without hurting users' function performance.

However, enabling such a system design is not easy: (i) The virtualization between cloud VMs and GPU devices is rarely studied before. GPU vendors such as NVIDIA provide proprietary solutions but lack flexibility [65]. This prompts us to implement a GPU virtualization mechanism from scratch and address issues such as virtual I/O, address translation, and resource isolation, which poses a challenge. (ii) GPU device re-mapping and model swapping can incur non-negligible latency. Reducing control-plane overhead and enabling user-transparent GPU sharing is challenging. (iii) Since model serving functions always exhibit different workload patterns and latency requirements, determining the optimal GPU allocation and request scheduling policy without violating the user's latency SLO is also a challenge.

To tackle these challenges, we present a cost-efficient GPU-enabled FaaS platform, `gShare`, that benefits both the user (i.e., model developer) and the cloud provider. `gShare` divides the GPU hardware into a set of different-sized GPU slices and allocates them to user functions through vGPU abstraction. The user can configure functions with a minimum of 128 MB of device memory and is encouraged to surrender GPU resources to receive discounts. In this mode, a user's idle vGPU instance can be dynamically mapped to other tenant function VM on the host server, with user-transparent checkpoint/restore on model data and an SLO-aware request scheduling algorithm, the GPU hardware can be fully utilized by different vGPU functions in a time-sharing manner, thus achieving high cost efficiency.

As one of the world's largest cloud providers, we use real-world workloads with extensive experiments to evaluate gShare's effectiveness and efficiency. Experiment results show it reduces GPU cost by 43%–63% while meeting 95% of users' latency targets compared to the keep-live policy, and outperforms the state-of-the-art method by 1.8×- 2.7× in both function performance and cost.

Our contributions can be summarized as follows.

- We reveal the inefficiency of existing serverless GPU functions based on several observations, and also give several key insights on the optimization of FaaS platforms to address this issue.
- We present gShare, which provides an elastic vGPU abstraction for a microVM-based FaaS environment. With a user-transparent vGPU switching and sharing technique, it enables flexible GPU allocation and brings many more opportunities for cost saving.
- We formulate the GPU sharing as a constrained optimization problem and explore the optimum design space. We also propose an efficient online scheduling algorithm that can significantly reduce GPU usage while satisfying users' latency SLOs.
- We compare gShare with several most related works, and the evaluations from real-world workloads demonstrate its high cost efficiency. For a typical FaaS cluster, it can help save hundreds of thousands of dollars in GPU server procurement costs every year.

# 2 Background & Motivation

## 2.1 Serverless Inference & GPU functions

Model serving is already widely used in a variety of areas. At Facebook alone, more than 200 trillion queries are served every day [46]. Maintaining these model-serving services at scale poses formidable challenges in terms of elasticity and resource costs, especially with unpredictable workload bursts. Serverless computing emerges as a compelling paradigm to address this problem, where ML models are encapsulated within lightweight FaaS containers (e.g., a security sandbox [28] or microVM [3]) and invoked by users on demand with rapid scalability. Nonetheless, model-serving applications are always latency-critical and computation-intensive, and early CPU-only FaaS functions often fall short of meeting users' low-latency requirements, resulting in unacceptable function performance.

GPU accelerators can significantly improve the model serving efficiency. For example, serving Resnet50 on the CPU can take hundreds of milliseconds, whereas on a GPU it completes in under 50ms [71]. Recognizing this potential, several FaaS providers have introduced GPU functions tailored for diverse model serving scenarios, which enable them to support a vast spectrum of low-latency ML applications, from simple image classification, speech recognition [5, 34, 84, 99], to large language models like LLaMA-2 and OPT [26]. The GPU function charges users based on both invocation count and resource consumption. At the same time, the resource pricing includes not only CPU and memory costs but also GPU-related expenses (depending on the hardware), which are derived by multiplying the execution time by the GPU quotas explicitly specified by the user.

## 2.2 Inefficiency of Existing FaaS platform

Despite the appeal of GPU capabilities, cost-efficient GPU management remains a challenge for FaaS providers. Our analysis reveals that current mainstream FaaS platforms exhibit notable inefficiencies in GPU operations.

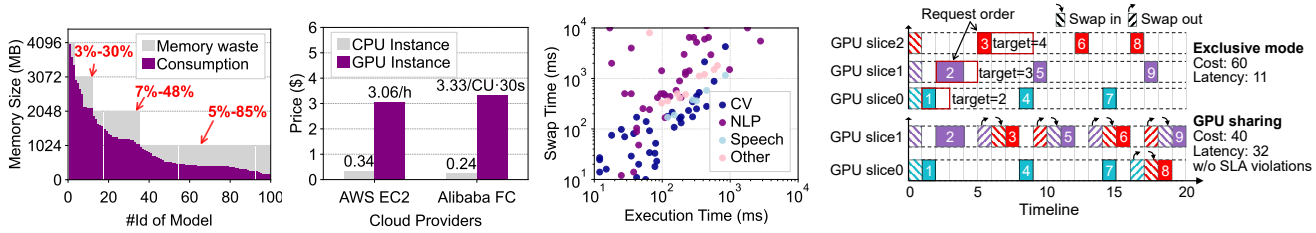
***Observation #1**: Coarse-grained GPU Allocation: The serverless GPU functions are inherently suitable for small-sized models but not effectively adapted to them currently, which can result in up to 85% of resource over-provisioning waste.*

Using FaaS functions to build highly elastic model-serving services has been proven to be a feasible solution, especially for small models that usually run on GPU fragments [99]. For large language models, deploying them via model slicing or parallelism also requires different types of GPU functions [26]. Modern FaaS platforms already support fine-grained CPU and memory allocation, allowing users to build small functions (e.g., a minimum of 0.05 vCPU cores with 128 MB of memory [15, 44]) to run a simple code block. While the shift towards GPU provisioning in FaaS environments is still underdeveloped, FaaS providers such as Alibaba GPU Function support GPU slicing at a granularity of 1GB of device memory, which is still insufficient.

We analyze the top 3,000 most downloaded models on TensorFlow Hub [38] and Hugging Face [24] (see § A) and estimate their memory footprint based on precision and parameter size. We find they exhibit a wide distribution of model sizes. More specifically, 78.1% of models are less than 4GB, while 66.4% are less than 1GB. We deploy several of them and present 100 model samples with memory consumption ranging from 128MB to 4GB. It can be seen that even using the smallest GPU function (1GB of device memory), there are about 60 functions that will suffer from a vast resource waste (with an average of 55% and a maximum of 85%). Similarly, about 20 models require at least 2GB of GPU memory for deployment, which can also lead to 7%-48% of GPU memory over-provisioning. This indicates that existing coarse-grained GPU functions may not precisely match user needs. Although GPU vendors such as NVIDIA offer proprietary virtualization solutions for VMs, the lack of flexibility and generality makes them unsuitable for dynamic serverless environments.

***Observation #2**: Keep-alive Cost: The existing keep-alive policy in FaaS platforms is ill-suited for GPU functions, leading to 10× of higher GPU-idling cost and poor resource utilization.*

Cold-start has been a well-known performance bottleneck in FaaS platforms. Launching a new function instance to




(a) Model memory footprint (b) GPU function price (c) Swapping & execution time (d) GPU sharing with SLO-aware request scheduling.

**Figure 1.** (a) GPU memory usage from common ML models. (b) The CPU price versus the GPU price in the cloud. (c) The model swap time and serving time among different models. (4) An example of efficient GPU sharing with feasible scheduling.

process a request when none are available often takes hundreds of milliseconds and may significantly impact function performance [22, 27, 98]. For GPU functions, the cold-start latency (usually in seconds) can be even higher than the model serving time due to the initialization of models, GPU drivers, and ML frameworks [34]. Function caching is widely used in FaaS platforms to avoid cold-start invocations. Most FaaS platforms choose to keep function instances alive for a period of time (e.g., up to 15 minutes in AWS Lambda [43]) after execution, so that subsequent requests can reuse them to eliminate cold-start overhead.

However, function caching also comes at a high cost. Unlike CPUs, which can be reclaimed by pausing idle instances [35], GPU resource management is coupled with main memory and cannot be released until the function is fully terminated. As shown in Figure 1(b), GPU instances such as AWS P3 (with NVIDIA V100 GPUs) are priced at approximately $3.06/h [74], compared to only $0.34/h for a t3.2xlarge CPU instance [73]. In Alibaba Cloud Function, GPU functions cost around $0.11/CU·s[1], which is approximately 14× more expensive than CPU functions at $0.008/CU·s [17]. Caching a large number of GPU function instances in a FaaS platform can lead to non-trivial cloud costs. As the workload variations can reach up to 33,000× within 1 minute in production systems [77], even using single-instance multi-concurrency techniques [37, 81], inflexible GPU allocation can also result in serious resource under-utilization, especially after handling bursty workloads [77]. This makes the keep-alive strategy unsuitable for GPU functions with higher unit prices.

***Observation #3**: Insufficient Over-selling: Some cloud providers utilize model swapping to reduce GPU costs, while the additional control plane overhead and conservative scheduling make them only bring limited benefit.*

To improve the cost-efficiency of GPU functions, recent FaaS providers offer a discounted billing model for idle function instances (i.e., preemptible GPU function), that is, users can allow the FaaS platform to swap their ML models from GPU device to host memory during the function's keep-alive period, thus the GPU resource can be reallocated to other tenants. For this goal, prior work deploys a GPU proxy on each worker node to manage all local GPU devices as a pool [103]. The GPU proxy creates an executor process on each GPU for remote CUDA API execution, model swapping, and device memory management. When a request arrives, the user function interacts with the GPU executor via a CPU client for model serving.

We find that the current model-swapping approach is inefficient for achieving high cost-efficiency in GPU functions. The GPU proxy is built on the FaaS control plane and handles request forwarding, memory swapping, and kernel processing when serving models. Such a two-layer GPU management design introduces additional system overhead and may increase the uncertainty in function performance. Although model swapping involves only memory copy operations and is faster than cold-start, it still incurs non-negligible latency (e.g., transferring 1GB of data using `cudaMemcpy()` takes approximately 1s). Moreover, ML models often exhibit inconsistent swap and execution times (see Figure 1(c)). Current systems swap models solely based on function popularity, which may underestimate the impact of model swaps on request latency and lead to frequent SLO violations. Conversely, overestimating swap overhead can limit resource utilization and hinder efficiency.

There is also the fact that cloud users often have diverse latency requirements for model serving queries. For instance, interactive services such as chatbots and real-time translation [29, 56, 76] typically require strict latency guarantees (e.g., <50ms per 100 tokens). In contrast, applications that rely on batch processing can tolerate longer request latency. Batching improves hardware utilization [5, 19] but also increases the makespan (e.g., serving Resnet50 with 8-batch input takes about 250ms). Furthermore, for GPU functions, users may opt for the GPU idle billing mode to reduce costs, indicating their willingness to accept potential latency increases or performance degradation. This provides an opportunity to explore more aggressive GPU sharing strategies. Figure 1(d) illustrates a scenario involving nine requests from 3 model serving functions with different latency targets (4, 3, and 2). With a carefully arranged request scheduling and

[1]CU (Computational Unit) is a billing model in Alibaba Cloud Function.

executing order, GPU resource usage can be reduced by approximately 30% in this case without violating users' latency SLOs.

### 2.3 Implications

The above observations motivate us to design a cost-efficient GPU-enabled FaaS platform based on the following key insight. (1) *Fine-grained GPU allocation*: A fine-grained GPU provisioning solution (e.g., time-slicing with vGPU) is essential for multi-tenant FaaS scenarios, enabling users to apply just the right amount of GPU resources for model serving functions, thereby minimizing over-provisioning waste (**Ob. #1**). (2) *Resource decoupling & Flexible sharing*: Leveraging the vGPU technology, we can implement a dynamic vGPU mapping mechanism to enable flexible GPU allocation and recycling, thus reducing the GPU idle rate. Meanwhile, optimizing the cost of GPU virtualization and the control plane on GPU management is necessary to reduce system overhead (**Ob. #2**). (3) *Efficient GPU overselling*: By co-designing the Leisure-aware vGPU allocation and SLO-aware request scheduling, we can explore an aggressive GPU overselling policy to achieve a balance between cloud cost in FaaS platform and users' performance requirements (e.g., latency targets), benefiting both the cloud providers and tenants (**Ob. #3**).

## 3 Methodology & System Design

In this section, we present the system design and core modules of gShare, along with the challenges encountered and the solutions we implemented.

### 3.1 System Architecture

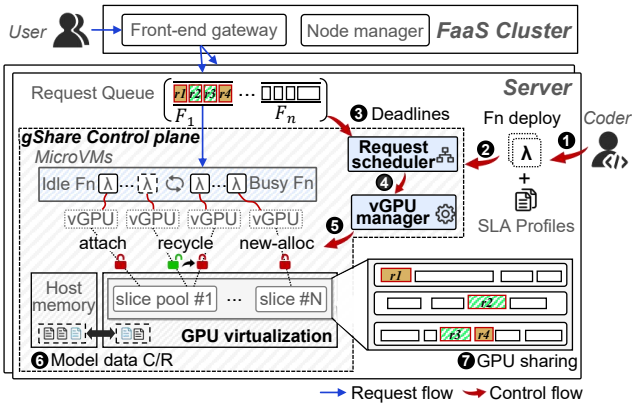



**Figure 2.** Overview of the system architecture.

**System Model**: Typically, a GPU-enabled FaaS cluster comprises multiple CPU servers and GPU servers. Upon invoking a GPU function, the request will be routed to a GPU server via a front-end gateway, based on a load-balancing policy. On each server, requests are first queued and then dispatched by a local scheduler to available GPU functions for execution. Although workloads may vary across servers, the underlying principle of GPU provisioning remains consistent. Therefore, we focus on general-purpose GPU function management architectures that are independent of cluster-level load balancing and can simplify the system design.

**Overview**: As illustrated in Figure 2, gShare is a server-level GPU overselling policy that aims to serve as many requests as possible from user functions, meet their latency SLOs while reducing the server GPU allocation. On each server, physical GPUs are allocated to user functions as vGPU instances, which share the GPU hardware in a time-slicing manner, with resource limitations on both device memory and computing capacity (e.g., 128MB of GPU memory and 10% of GPU time quota). gShare encapsulates these vGPU instances as different-sized GPU slices, determines their mappings to the user functions dynamically based on system status and workload changes through three key components: (1) a *GPU virtualization* mechanism that enables fine-grained vGPU provisioning in a multi-tenant FaaS environment, where user's ML model services run in microVM-based containers; (2) a newly introduced *vGPU manager* that unlocks GPU resource from function pausing and provides interfaces for vGPU mapping and model saving/restoring; (3) a modified *request scheduler* that determines request execution order and vGPU allocation for efficient GPU sharing.

gShare operates as a back-end module on each server. For developers, gShare allows user to explicitly specify their vGPU quotas and latency SLOs, and also provides an automatic tool to help infer the model's memory consumption (including the parameter sizes, libraries, and intermediate data) ❶. When a model serving function is deployed, gShare collects its profiling metadata, such as the function name, model size, and average execution time, which can be analyzed from the native logging system or observability modules in the FaaS platform. The profiling can be done only once❷. When user requests arrive, the *request scheduler* estimates the deadline slack of queued requests❸, and determines whether to reuse an already assigned GPU slice or allocate a new one❹. Subsequently, it invokes the *vGPU manager* to map the function's vGPU to the corresponding GPU slice pool❺ and load the model data to the device memory for processing❻. Through fine-grained vGPU allocation and flexible scheduling, gShare enables efficient GPU sharing across multiple tenant functions❼.

The following sections will elaborate on the detailed design, methodology, and implementation of gShare.

### 3.2 VM-passthrough GPU Virtualization

Previous studies primarily implement GPU virtualization via user-space API interception in the guest VM [79, 100]. These methods are easy to implement but lack generality, rendering them tied to specific frameworks or ML stacks. Since users' custom images in the public cloud often require

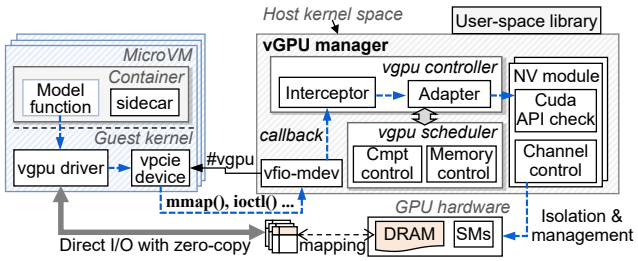



**Figure 3.** The GPU virtualization design.

modifications to the VM's dynamic libraries, we use kernel-space interception to virtualize GPU devices. Although this approach is challenging to implement and rarely studied, it can provide better compatibility.

Figure 3 shows the architecture of our vGPU design. gShare implements a core component, named *vGPU Manager*, to provide vGPU devices for user functions on each server, which hides the details of physical GPU slices and exposes a set of vGPU driver APIs for user applications inside microVM. The *vGPU manager* is launched as a kernel process within the host operating system and incorporates two key modules: *vgpu controller* and *vgpu scheduler*. (1) The *vgpu controller* includes a series of I/O control interfaces that are adaptive to different GPU vendor devices; it intercepts the system calls from user functions for API filtering, data packet parsing, and permission verification. (2) The *vgpu scheduler* manages resource quotas, performs kernel-level task switching, and eventually executes requests by calling the GPU hardware driver on the host server. In the following, we elaborate on the design details of vGPU functions.

**vGPU Allocation**: When launching a new GPU function, a vGPU instance will be created and attached to the VM, and the user's resource quota will be recorded in the kernel object on the host server. The *vGPU manager* maintains this information and limits the resource usage from runtime checks. gShare theoretically supports arbitrary granularity of GPU function creation; however, for practical manageability, we allow users to configure GPU functions in units of 128 MB of device memory, and the computation resources (i.e., GPU time) are proportionally limited. For the FaaS platform, the *vGPU manager* also exposes a user-space library to facilitate vGPU device management, such as creation and deletion operations, where Table 1 lists a subset of them. These user-space APIs can be seamlessly integrated with the third-party programs, libraries, or scheduling modules on the host server, ensuring compatibility with orchestration frameworks such as Kubernetes [42] and Slurm [87].

**Direct I/O Access**: gShare leverages vfio-mdev [53], a Linux kernel feature that enables direct I/O device access between hardware and virtual machines via technologies such as Intel VT-d or AMD-Vi, thereby minimizing virtualization overhead. Upon creating a vGPU instance, vfio makes a virtual I/O device within each VM, allowing the guest process

**Table 1.** A subset of the vGPU user-space library.

| Interfaces | Parameters |
|---|---|
| create_vgpu() | vgpu_config, vgpu_mem_address*, tenant_id |
| delete_vgpu() | vgpu_mem_address* |
| get_vgpu_status() | vgpu_mem_address* |
| set_vgpu_config() | vgpu_mem_address*, vgpu_config |

to access device memory and registers on the host server via standard system calls (e.g., vfio_pci). A complete set of vGPU driver interfaces within the guest VM is implemented to fully support various mainstream model-serving frameworks. When the vGPU device works, the *vgpu controller* monitors the vfio API callbacks on the host server from its I/O control interfaces, thereby intercepting and handling the guest-VM's GPU kernel invocations.

**vGPU Scheduling**: The *vgpu scheduler* is responsible for managing hardware resources and consists of a vGPU memory control and computation unit control. For example, in the case of NVIDIA GPUs, it captures the CUDA driver API intercepted by the *vgpu controller* and evaluates memory-related API calls to enforce vGPU memory usage limits. Regarding SM cores, the scheduler implements temporal-sharing among multiple vGPU kernels. Given that the hardware scheduler in NVIDIA GPUs only retrieves commands from the PushBuffer of an active channel [62], the *vgpu scheduler* maintains a record of the corresponding channel for each vGPU and ensures that only one channel is active at any time. Similar to NVIDIA's closed-source solution [63], we employ a round-robin algorithm to schedule user functions' GPU tasks in a configurable scheduling period (20ms by default).In each period, kernels from different vGPU instances execute in FCFS (First-Come-First-Served) order according to their vGPU instance creation timestamps. Each kernel exclusively occupies the GPU device within its time slice ($slice = func\_mem/dev\_cap \times sche\_period$) and cannot be preempted even if its cycle is idle.

### 3.3 Function Memory Management

The number of vGPU instances that can be created on a physical GPU is limited by the device memory capacity. Therefore, recycling idle vGPU instances (i.e., GPU overselling) can help accommodate more tenant functions on the GPU hardware, thereby reducing GPU usage in the FaaS cluster. For this goal, we further develop efficient mechanisms for vGPU hot-plugging and model swapping to enable dynamic GPU

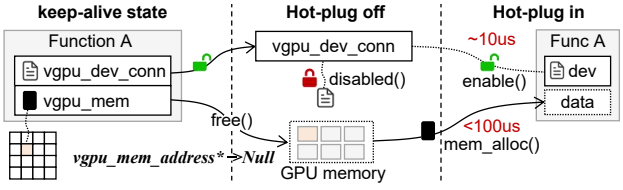



**Figure 4.** The vGPU hot-plugging mechanism.

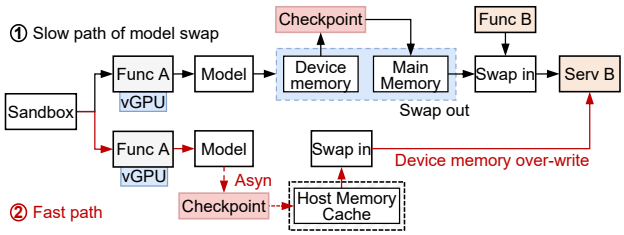



**Figure 5.** The fast model swapping mechanism.

offloading and reallocation across tenants. The detailed technical implementation is described in the following sections.

**vGPU Hot-Plugging**: The hot-plugging for PCIe devices is natively supported in the vfio-based virtualization technique. However, this operation inevitably introduces latency. For example, inserting a GPU device into a VM via PCIe typically takes approximately 0.7s, significantly hindering the efficiency of GPU resource recovery and reallocation. To mitigate this overhead, we implement a "pseudo offloading" strategy (Figure 4), that is, when a GPU slice is being unloaded from an idle tenant function, only the hardware resource is released, while the associated vGPU connection, including file descriptors, process handlers, and register information is tagged as disabled from using. Upon resuming the tenant function, the *vGPU manager* only needs to reactivate the GPU slice without instantiating a new vGPU device. In this way, the vGPU hot-plugging time can be reduced to less than 1ms.

**Checkpoint & Restore**: As the model parsing and loading often take considerable time, it is crucial to avoid redundant initialization operations for vGPU functions. Therefore, for functions in the keep-alive state, their model data and state should be saved before unloading the GPU device. We utilize the CUDA checkpoint to create a model snapshot for GPU functions [59, 82]. Unlike conventional proxy-based approaches that require a GPU server to participate in the model C/R process, we extend microVMs' sidecar functionality to support direct model data management within GPU functions. The FaaS platform on the host server initiates the C/R operation externally. By default, model checkpoints consume private memory in the user's own function VMs. Since tenant functions may not have sufficient idle memory for snapshot storage, we build a shared memory pool on each server and map it to tenant VMs, enabling model recovery in memory rather than reading checkpoint files from disk. The memory pool is limited by the available memory in the FaaS cluster, and an LRU cache replacement policy is used when the pool is exhausted.

**Fast Model Swap**: In prior systems, reallocating GPU resources from one tenant to another typically involves a pair of model swap-out and swap-in operations [103]. We introduce a GPU memory "overwrite" mechanism based on snapshot cache technique to reduce the model swapping time. As illustrated in Figure 5, when a function's model (function $A$) is checkpointed for the first time (or in an asynchronous way), it will be cached in main memory. After that, if its GPU device is allocated to another function, only a model swap-in operation (function $B$) needs to be done, and the model swap-out operations of function $A$ can be skipped. The black-box design of the NVIDIA C/R tool prevents us from remapping GPU memory addresses at the current stage, and performing GPU memory overwrites requires terminating function $A$'s user process.

### 3.4 SLO-aware Request Scheduling

To facilitate management, we redefine the function instance states within the FaaS cluster. On each server, a function instance can be in one of three states: running, idle, or frozen. A newly created function instance immediately enters the idle state. When it executes a request, it transitions to the running state. Upon request completion, the instance transitions from running to idle, retaining all resources and runtime data (e.g., the memory heap and register data). Instances that remain idle for an extended period may transition to the frozen state, where the process is suspended, memory resources are preserved, and CPU and GPU resources can be deallocated and reassigned to other functions. By reallocating CPU and GPU resources, a frozen function instance can be restored to the idle state.

The incoming requests are first queued in a waiting buffer, and `gShare` must process them before their respective deadlines while minimizing GPU resource allocation. For each request, the *request scheduler* has two options: (1) wait for a running function instance to become idle, reuse its GPU slice, and load the corresponding model data for execution; or (2) unpause a frozen instance or launch a new instance, allocate a new GPU slice, and process the request. The former approach reduces GPU resource consumption but may introduce request queuing delays, while the latter enables faster execution at the cost of higher resource usage. We assume that all arriving requests have at least one pre-launched function instance and do not consider occasional cold-start events. The scheduling objective of `gShare` is to dynamically determine on which GPU time slice to execute which requests under a given workload (request arrival) and server resources, thereby serving as many user requests as possible while meeting the SLO latency target of the requests, that is, achieving a high GPU over-selling rate.

#### 3.4.1 Problem Formulation.

The joint optimization of GPU allocation and request scheduling resembles an online bin packing problem with per-request deadline constraints. Since request arrivals follow a discrete time-series distribution, `gShare` works in a cycle-by-cycle fashion, numbered by $t = 1, 2, \ldots$, and we denote by $R^t$ the set of requests arriving in cycle $t$. For brevity, we may omit the time subscript/superscript $t$ for certain variables in the following formulation. Let $M$ denote the maximum capacity of the GPU slice pool, and each GPU slice is divided into $K$ time slots, where $K$ is a

large integer, and the time length of each slot is a fixed value $\Delta t$. For each request $r_k \in R$, we assume knowledge of its function profiling data, including model swap time $I_k$, execution time $D_k$, arrival timestamp $K_k$, and latency requirement $\theta_k$. The objective is to determine a binary decision variable $a_{j,k}(\delta)$, which equals "1" if the $k$-th request $r_k$ is scheduled at time slot $\delta$ (where $\delta \in t$) on the $j$-th GPU slice, and "0" otherwise.

This optimization problem can be formulated as follows.

$$minimize: \quad \sum_{j\in M} U_j \tag{1}$$

$$\sum_{j\in M,\delta\in t} a_{j,k}(\delta)\delta \geq \sum_{j\in M,\delta\in t} a_{j,k}(\delta)K_k, \quad \forall r_k \tag{2}$$

$$\sum_{j\in M,\delta\in t} a_{j,k}(\delta) = 1, \quad \forall r_k \tag{3}$$

$$\sum_{r_u,r_v\in R^t} s^j_{u,v} = \sum_{r_k,\delta\in t} a_{j,k}(\delta), \quad \forall j \tag{4}$$

$$s^j_{u,v} + s^j_{v,u} = 1, \quad \forall r_u, r_v, \forall j \tag{5}$$

$$\sum_{r_u\in R^t} s^j_{u,v} \leq 1, \quad \forall r_v, \forall j \tag{6}$$

$$\sum_{r_v\in R^t} s^j_{u,v} \leq 1, \quad \forall r_u, \forall j \tag{7}$$

$$\sum_{\delta\in t} a_{j,v}(\delta)\delta \geq \sum_{\delta\in t} a_{j,u}(\delta)\delta + D_u - H(1 - s^j_{u,v}), \quad \forall r_u, r_v, \forall j \tag{8}$$

$$S_k\Delta t + I_k(1 - F_{k',k}) + D_k \leq \theta_k, \quad \forall r_k \tag{9}$$

$$0 \leq \sum_{j\in M,r_k} a_{j,k}(\delta) \leq M, \quad \forall \delta \in t \tag{10}$$

The objective (1) defines the overall GPU resource cost on active function instances, where $U_j = 1$ if GPU slice $j$ executes at least one request during cycle $t$ (i.e., $\sum_{\delta\in t,r_k\in R^t} a_{j,k}(\delta) > 0$), and $U_j = 0$ otherwise. Constraint (2) ensures that no request can be scheduled before its arrival timestamp. Constraint (3) guarantees that each request is executed exactly once on a single GPU slice. Constraints (4–7) enforce a strict execution order between any two adjacent requests $r_u$ and $r_v$ assigned to the same GPU slice. A binary variable $s^j_{u,v}$ indicates the scheduling sequence: $s^j_{u,v} = 1$ if both $r_u$ and $r_v$ are assigned to the same GPU slice $j$, and $r_u$ is scheduled immediately before $r_v$; otherwise, $s^j_{u,v} = 0$. Constraint (8) ensures that there is no overlapping execution time between consecutive requests on the same GPU slice. Constraint (9) imposes that the summary of a request's waiting time and model swap time (if it has one) must not exceed its latency target. Here, $S_k = \sum_{j\in M,\delta\in t} a_{j,k}(\delta)(\delta - 1)$ represents the start time of request $r_k$, and $F_{k',k} = 1$ if requests $r'_k$ and $r_k$ belong to the same tenant function, and otherwise, $F_{k',k} = 0$. Constraint (10) ensures that the total number of allocated GPU slices does not exceed the maximum capacity of the GPU slice pool at any time slot $\delta$. Due to the page limitation, we put more details in § B.

This MINLP problem includes both the spatial-dimension function-slice mapping and temporal-dimension time-slot exclusivity with strict request execution order assignment, which is at least as hard as the known NP-hard bin packing problem [72]. Therefore, we resort to a fast algorithm for efficient scheduling in practice. Algorithm 1 outlines the detailed procedure. We utilize a minimum heap data structure to maintain the requests awaiting scheduling and the already

**Algorithm 1:** Dual-Queue Lazy Scheduling (DQLS)

```
1  Init shareQueue by request deadline slack in ascending order
2  Init slicePool by slice's next available time in asc-order
3  while true do
4      R_t ← recvNewArrivedReq() ;
5      shareQueue, cacheQueue ← classifyByFunc(R_t) ;
6      update the slicePool on the server ;
7      while cacheQueue ≠ ∅ do
8          r_k ← cacheQueue.pop() ;
9          cacheHit ← findAvailCachedSlice(slicePool) ;
10         if cacheHit then
11             slice ← slicePool.pop() ;
12         else
13             slice ← allocGpuSlice() ;
14         slice.process(r_k) ;
15     while slackQueue ≠ ∅ do
16         r_k ← shareQueue.pop(); // pick request with the smallest slack
17         if r_k.slack = 0 then
18             slice ← findAvailSharedSlice(r_k) ;
19             if slice = Null then
20                 slice ← allocGpuSlice() ;
21             slice.process(r_k) ;
22         else
23             if idleSliceExists(slicePool) then
24                 processWithIdleSlice(r_k) ;
25             else
26                 break ;
27     for r_k ← slackQueue do
28         r_k.slack ← r_k.slack − 1 ;
29 Function findAvailSharedSlice(r_k):
30     max ← −1; slice ← Null ;
31     for s ∈ slicePool do
32         ddlLeft ← estimation(s, r_k) ;
33         // predict the request completion time
34         if ddlLeft ≥ 0 & max < ddlLeft then
35             max ← ddlLeft, slice ← s ;
36     return slice ;
```

allocated GPU slices in `shareQueue` and `slicePool`, respectively (Lines 1–2). Requests in `shareQueue` are prioritized based on their deadline slacks, defined as $slack = \theta_k - I_k - \hat{D}_k$, where $\hat{D}_k$ denotes the predicted execution time of request $r_k$, derived from the historical average latency (as discussed in § 5.2). The *function scheduler* continuously monitors incoming requests, categorizes them into two groups according to the risk of latency target violation (Lines 3–5), and updates the pool of allocated GPU slices (Line 6). If the estimated total time $I_k + \hat{D}_k$ exceeds the latency target $\theta_k$, indicating a high risk of SLO violation, the *function scheduler* opts to cache the corresponding function instance individually rather than sharing its GPU slice (i.e., placing it into `cacheQueue`). These requests are then processed using a locality-aware scheduling to minimize the frequency of model swapping (Lines 7–14).

A lazy scheduling is applied to the `shareQueue`. A request is scheduled only when its deadline is being violated (Lines 15–27). This approach provides an opportunity to reprioritize the order of request execution to improve vGPU

utilization. For the request with the smallest *slack* value (Line 16), the following scheduling decisions are made:

(1) If any request currently has $slack = 0$, the *function scheduler* predicts the waiting time (queuing and swapping) and execution time, and selects the GPU slice with the earliest available time for scheduling. If no such slice is available, a new GPU slice is allocated (Lines 18–22).

(2) When all scheduled requests maintain a safe slack margin ($slack > 0$), the *function scheduler* still allows the request with the smallest *slack* to be assigned to an idle GPU slice, thereby reducing resource idling and improving overall efficiency (Lines 23–27).

Lines 30–37 show the `findAvailSharedSlice()`. Since certain GPU slices may still have ongoing requests, the *function scheduler* must estimate their next available time. The algorithm computes the expected completion time on each GPU slice by incorporating model swap times and latency constraints on incoming requests. If no allocated GPU slice can be reused, the scheduler is notified to allocate a new GPU slice by calling the `allocGpuSlice()`. The maximum number of GPU slices that can be allocated on a server depends on the hardware capacity. If `allocGpuSlice()` fails, the request will be redirected to other nodes for processing.

## 4 Implementation

**Runtime & Hardware Adaption**: `gShare` is integrated into a microVM-based FaaS platform equipped with NVIDIA GPUs, where GPU functions are encapsulated within Kata Containers [69] running on bare-metal servers. We chose NVIDIA as our first choice because its broader ecosystem, and users with NVIDIA GPU requirements account for 95% in our platform. The kernel-space API interception on vGPU drivers requires us to rewrite all system calls. We have rewritten a total of 57 ioctl interfaces (17 NV_ESC APIs and 40 UVM_APIs), including 60,000 lines of C code on vGPU design, state management, and unit testing. The current version of `gShare` provides a full CUDA interface and is compatible with today's mainstream ML frameworks, such as TensorFlow Serving, PyTorch, vLLM, and SGLang. Adaptation efforts for AMD ROCm [2] are in progress.

**Components & Prototype**: The *request scheduler* is a native module within the server Agent on each node that operates as a background daemon responsible for request routing, function management, and node status monitoring. We extend the server Agent to incorporate model C/R and function profiling mechanisms. The input data used by `gShare` for decision-making is collected from the logging system and stored in a cluster-level repository. A prototype system is also developed to evaluate the effectiveness of `gShare`, which includes the main modules with the same settings as the production system. There are about 5,000 lines of code (Java, Python, and Linux Shells) for the functional verification, workload generation, and experimental result processing.

## 5 Evaluation

We evaluate `gShare` in a testbed comprising 20 physical servers, each equipped with 128 GB of RAM and a 32-core CPU. The cluster hosts 64 NVIDIA A100 GPUs (40GB of device memory), divided into 9 GPU slice pools ranging from 128MB to 40 GB of vGPU types. We use MLPerf [70] and generate workloads following real-world traces from three production clusters, each containing one week of function calls with diverse request arrival patterns and function characteristics. By default, we define a function's latency target as 1.5 × its 90th percentile (i.e., $p90$) execution time.

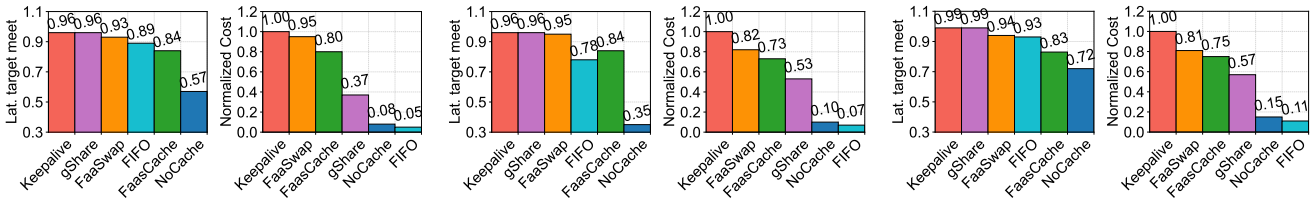
We compare `gShare` against five related systems: Keepalive, FaasCache, FaaSwap, NoCache, and FIFO. The detailed implementation of these works is listed as follows:

- Keepalive [43] is a widely used function caching mechanism in current FaaS platforms, which employs a TTL-based (time-to-live) keepalive policy to cache function instances in memory after each invocation. We adopt AWS Lambda's 15-minute idle timeout setting for GPU functions as a baseline.
- FaasCache [27] is a state-of-the-art method in function caching, which proposes a priority-based cache policy for CPU functions. We transfer it to the GPU functions and swap models based on function popularity, model size, and swapping overhead.
- FaaSwap [103] is also a state-of-the-art method in GPU function sharing[2] It implements a model-swapping mechanism and determines the request execution order using feedback control with an LRU cache policy. Since it is not open source, we re-implement its scheduling algorithm for comparison.
- NoCache is an aggressive strategy that immediately releases vGPU resources when a function becomes idle. We enhance it with a snapshot-based cold-start mechanism to reduce cold-start overhead while minimizing resource consumption.
- FIFO is a homogeneous version of `gShare`, which employs a single *slackQueue* and early scheduling to allocate vGPU resources immediately upon request arrival, without considering the user's latency deadline slack.

In the evaluation, we mainly focus on the function performance and resource cost around the following questions:

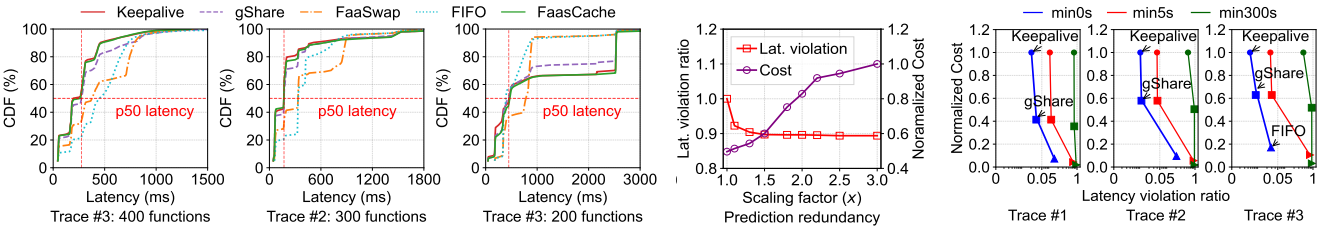
- Q-1: Can `gShare` achieve better cost-efficiency under different workloads? (§ 5.1)
- Q-2: How does `gShare`'s GPU sharing mechanism affect request latency? (§ 5.2)
- Q-3: How does `gShare` perform under various function concurrency and latency constraints? (§ 5.3)
- Q-4: What is the system overhead of `gShare` on vGPU allocation and reclamation? (§ 5.4)

[2]FaaSwap is an early manuscript version of Torpor[104]. They are equivalent in terms of references.




(a) Trace #1: 400 functions, QPS<5 (b) Trace #2: 300 functions ,QPS=9.6 (c) Trace #3: 200 functions, QPS>10

**Figure 6.** Overall performance under different workloads: (a) a low concurrency workload including model serving requests from 400 functions, with a less than five queries per second (QPS); (b) a medium-scale workload with 300 functions and QPS=9.6; (c) a high concurrency workload with 200 functions and QPS>10. Experimental results are normalized to baseline.




(a) CDF of request latency distribution on gShare (b) Parameter tunning (c) Pareto optimal scheduling on gShare

**Figure 7.** (a) Latency distribution of bad requests. (b) Parameter tuning of gShare. (c) The normalized cost and function performance of Keepalive, gShare, and FIFO under different workload patterns, where the latency target setting is in default.

- Q-5: What is the performance roofline of gShare and how much economic benefit can it bring?(§ 5.5 and § 5.6)

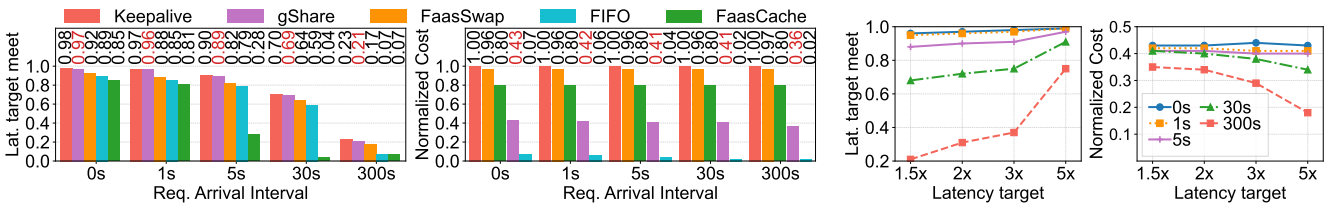
### 5.1 Overall Performance

**High Efficiency: gShare reduces GPU usage by 43%–63% while maintaining function performance comparable to baseline.** Figure 6 presents the overall performance of gShare. We collect experimental results at two-hour intervals (i.e., cycle $t = 2h$). The peak GPU allocation within each interval is aggregated to the overall cost, while all requests are aggregated to compute the overall latency target violation ratio. We see that the Keepalive delivers the best function performance but incurs the highest GPU resource cost. Compared to the baseline, gShare exhibits only a slight performance degradation while achieving significant cost reductions of 63%, 47%, and 43% across the three evaluated workloads, respectively. Although FaaSwap reduces GPU usage by 5%–20%, it prioritizes scheduling requests from functions with the highest latency violation ratios. This feedback-driven model swapping performs insufficiently, resulting in 1.75×–6× more latency violations. A similar behavior can be seen in FaasCache. The early scheduling mechanism in FIFO minimizes resource usage but causes higher latency target violations, particularly for functions with long model swap times. Moreover, the NoCache strategy is overly aggressive, where frequent model swaps severely degrade function performance.

**Workload Agnosticism: gShare delivers consistent performance improvements across diverse workload scenarios.** The experimental results in Figure 6 also demonstrate that gShare is workload-agnostic. Despite variations in tenant count, request arrival intervals, and function concurrency across the three workloads, gShare maintains robust performance. For example, it outperforms Keepalive, FaaSwap, and FaasCache in terms of cloud cost, while achieving better function performance compared to FaaSwap, FIFO, FaasCache, and NoCache. Even on a single server, functions can have multiple independent and concurrent invocations. If FaasCache's cache replacement policy were applied to determine model swapping priorities, system efficiency would heavily depend on workload characteristics such as function popularity and model size, and lead to frequent performance fluctuations (Figure 6(b)). gShare's server-level scheduling mechanism takes function profiling data as input and actively participates in resource allocation and request execution, enabling it to adapt to various ML scenarios.

### 5.2 Request Latency Distribution

**Latency Impact: The lazy scheduling mechanism in gShare may increase request waiting time, but only results in a 15% increase in $p95$ latency, with negligible degradation in $p50$ latency.** We analyze the latency distribution of latency target-violated requests (called "bad requests") from the experiments described in § 5.1. Note that




(a) Comparison under different function concurrency on Trace #1

(b) Comparison under various latency targets on Trace #1

**Figure 8.** A detailed analysis on latency target meet ratio and resource cost of gShare, where we take Trace #1 as an example and present gShare's performance under different function concurrency (a) and latency target settings (b).

**Table 2.** Latency breakdown of five compared methods.

| Method | #Requests meet SLOs | Request latency breakdown (①②) Latency on wait① | Latency on swap② | #Requests violate SLOs | SLO violation reasons (①②③) #by wait① | #by swap② | #by exec③ |
|---|---|---|---|---|---|---|---|
| Keepalive | 201,578 | 0.07% | 0.16% | 4,251 | 7.88% | 92.12% | -/- |
| gShare | 199,390 | 0.07% | 0.82% | 6,439 | 5.36% | 94.31% | 0.33% |
| FaasSwap | 183,227 | 0.15% | 1.02% | 22,602 | 2.96% | 95.44% | 1.60% |
| FIFO | 189,969 | 0.54% | 0.25% | 15,860 | 12.6% | 84.93% | 2.50% |
| FaasCache | 175,989 | 0.07% | 0.11% | 29,840 | 8.31% | 91.69% | -/- |

bad requests are inevitable in our evaluation, as some functions are invoked only once, and their initialization time alone may exceed the latency target. Additionally, prediction errors in gShare's scheduling algorithm can also lead to latency increase and performance fluctuation. Nonetheless, as shown in Figure 7(a), severe performance degradation in gShare is rare. Specifically, the $p50$ latency of gShare is comparable to or better than that of other methods. For $p90$ latency, gShare exhibits a slight increase and occasionally performs worse than FaasCache. However, the performance gap narrows progressively from 60% at $p90$ to 15% at $p95$ latency and $p99$ latency. This indicates an expected latency impact of gShare. As illustrated in Figure 7(b), if we add 30%-50% redundancy to the original predicted model execution time (i.e., the average latency) in gShare, the performance degradation caused by prediction errors can be obviously improved, and we set the redundancy at 40% in practice.

### 5.3 Evaluation on Case Studies

We also evaluate the effectiveness of gShare under varying request concurrency and latency constraints by changing the minimum request arrival interval (*0s* (default), *1s*, *5s*, *30s*, and *300s*). We also use three additional latency targets as 2×, 3×, and 5× the $p90$ execution time.

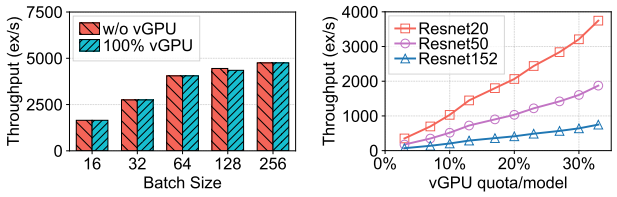
**Pareto Optimality: gShare achieves Pareto optimality on cost-efficiency under different workload settings.** We evaluate gShare under 20 different combinations of function concurrency and latency target. The results show it outperforms FaaSwap and FaasCache in most cases and achieves Pareto optimality among Keepalive, FIFO, and NoCache. A subset of the evaluation results is presented in Figure 7(c). As shown, gShare strikes a favorable balance between resource cost and function performance. In contrast, Keepalive and FIFO exhibit extreme scheduling behaviors, either incurring high costs or causing numerous latency target violations.

**Sensitivity Analysis: gShare performs best under sparse workloads with lenient latency targets.** Given gShare's workload-agnostic design, we use trace #1 as an example for detailed case studies. Figure 8(a) presents the evaluation results with the latency target fixed at the default. We observe that gShare's function performance is only 5% lower than Keepalive but significantly better than other methods across all groups. Moreover, it reduces resource usage by approximately 60% and 50% compared to the baseline and the state-of-the-art method, respectively. It is evident that gShare performs particularly well in environments with a large number of tenant functions and sparse request arrival patterns. In contrast, using Keepalive alone leads to significant resource waste in such scenarios. It is worth noting that gShare experiences a noticeable performance drop from the *min5s* to *min300s* groups. This is primarily due to the increased cold-start ratio caused by filtering out requests to simulate sparse workloads, as demonstrated in Figure 8(b), relaxing the latency target can further improve the function performance of gShare.

To help understand the root causes of latency target violations in gShare, we present the latency breakdown of gShare in Table 2. The left three columns focus on user requests that meet their latency targets, primarily presenting the number of requests that satisfy latency targets under each method, as well as the number of requests that have experienced at least one scheduling wait or model swapping event. It can be seen that gshare's swapping and scheduling waiting time has a <1% impact on preventing the requests from meeting their targets. Conversely, the right three columns focus on user requests that violate latency targets, analyzing

**Table 3.** GPU price and memory price in AWS EC2 and Aliyun ECS.

| | Instance types | GPU devices | $/Instance (per GPU) | Memory price |
|---|---|---|---|---|
| AWS EC2 | p4d.24xlarge<br>c5.xlarge | NVIDIA A100×8<br>-/- | $21.96/h ($2.74/h)<br>-/- | -/-<br>$0.013/GB/h |
| Aliyun ECS | gn8is-2x.8xlarge<br>g7.xlarge | NVIDIA L20×2<br>-/- | $4.53/h ($2.27/h)<br>-/- | -/-<br>$0.0069/GB/h |




(a) Performance loss of vGPUs (b) Resource isolation on vGPUs

**Figure 9.** Function performance with GPU virtualization.

the proportions of scheduling wait time, model swapping time, and execution phase in the overall request latency (the latency of all requests is summed). We find that model swapping time is the primary cause of latency target violations across almost all methods, underscoring the need to profile model swapping time. Furthermore, gShare's scheduling latency is slightly higher than that of FaaSwap, but it produces fewer latency target violations compared to FaaSwap, FaaSCache, and FIFO methods, further demonstrating the effectiveness of SLA-aware scheduling.

### 5.4 Virtualization Performance & Isolation

We also evaluate the performance overhead introduced by virtualization as well as the control plane efficiency.

**Virtualization Overhead: gShare's vGPU incurs negligible performance degradation for microVM-based functions.** The fine-grained GPU provision and sharing in gShare incorporate several optimizations, such as the memory zero-copy technique and native scheduling on GPU channels to minimize the system overhead. We first analyze the performance impact from GPU virtualization, which primarily stems from instruction delivery and memory address translation during guest-host driver API calls. Figure 9(a) compares the model throughput on a full physical GPU versus a 100% quota of vGPU under varying input sizes. There is virtually no performance loss between them.

**Resource Isolation: gShare ensures strong resource isolation across multiple tenant functions sharing the same GPU device.** Effective resource isolation is essential in a multi-tenant environment, as both resource contention and context switching can degrade function performance. To evaluate this, we deploy several ML models such as BERT, RCNN, Resnet, SSD, and VGG, and each of them runs on different GPU slices from the same physical GPU device. We take the image recognition models (Resnet20, Resnet50, and Resnet152) as examples and show their inference performance when serving concurrent requests. Figure 9(b) illustrates the function throughput under different vGPU quotas that are equally allocated to each tenant (ranging from 3% to 33%). The results show a nearly linear relationship between vGPU allocation and model throughput, while the performance of each vGPU function remains stable when processing concurrent requests from different users. This also demonstrates gShare's robust vGPU isolation capability.

### 5.5 Cost Benefit Analysis

**Economic Benefit: gShare can help reduce GPU resource cost by hundreds of thousands of dollars every year in a typical FaaS cluster, benefiting both cloud provider and users.** As the cost between host memory and GPU devices may vary across cloud providers, we estimate cost savings based on AWS EC2 GPU pricing [74] and Aliyun. As shown in Table 3, we reference the price of the AWS p4d.24xlarge instance, where each GPU costs approximately $2.74 per hour. Cloud providers do not provide a separate price for main memory; therefore, we estimate it based on the formula: $p_{total} = P_{cpu} + P_{mem} + P_{gpu} + P_{other}$, where $P_{other}$ represents the network/management overhead. For example, the memory price of the AWS c5xlarge series is approximately $0.013/GB/h. Considering a typical FaaS cluster usually contains 200–300 servers, where at least 10% of them are equipped with 8 A100 GPUs per server, the monthly GPU server rental cost is no less than $2.74 × 20 × 8 × 720 × = $315,648.

By adopting gShare's GPU sharing policy, GPU resource usage in a FaaS cluster can be reduced by up to 40% even in the worst-case scenario. If 20% of planned GPU server procurement can be cut, the monthly GPU cost for the FaaS provider will drop to approximately $252,000. Additionally, model swapping consumes more than 30% of GPU server memory, incurring additional memory costs of $0.013 × 20 × 8 × 24 × 720 = $35,942.4, for savings of about $330,000 per cluster per year. The cost benefit in Aliyun can be calculated in the same way, since it has cheaper main memory than AWS, the cost saving can reach up to about $400,000/year when using the Aliyun pricing model.

### 5.6 Discussion

**The Optimal Solution**: gShare achieves 75% of the optimal cost-efficiency but being 100× faster in practice. To explore the roofline of gShare's scheduling algorithm, we compare

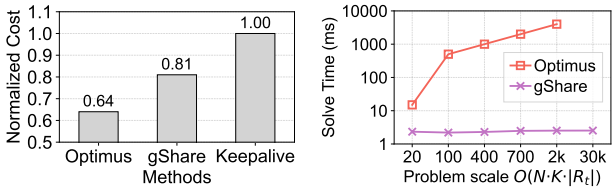
**Table 4.** Comparison of some representative related work.

| Feature \ Method | MArk | Nexus | Clockwork | INFless | FaaSwap | gShare |
|---|---|---|---|---|---|---|
| GPU sharing | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| Fine-grained alloction | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ |
| vGPU support | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| SLO-aware scheduling | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| Model swap | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| FaaS scenario | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |

it with the optimal solution solved by Lingo 18.0.44 on a small-scale workload. To simplify the problem, we assume a large GPU slice pool and define the latency target as 5× the $p90$ execution time, which is greater than the cold-start time to ensure the latency SLO is guaranteed during scheduling. We scale the workload timeline by a factor of 1:40 and set the number of time slots per GPU slice to 50 (i.e., scheduling every 2000ms). As shown in Figure 10(a), gShare's online scheduling algorithm achieves 75% of the optimal solution, while its decision-making speed is 10×–100× faster than that of the MINLP solver. With the increase of problem scale, Lingo's solving time grows exponentially and becomes infeasible, as illustrated in Figure 10(b).

**Temporal-Spatial sharing**: GPU virtualization is actually a temporal GPU sharing technology in gShare, which executes the GPU kernels from the user VM in a non-preemptive manner in each time slice. This may inherently cause underutilization of resources if a kernel cannot launch enough threads to occupy all SM cores on the GPU device [1, 20]. Fortunately, the vGPU technique can be combined with MIG technology to achieve approximate "spatial-temporal sharing" on NVIDIA GPUs (partially supported). In this way, the GPU is first divided into small physical partitions (e.g., 5GB of device memory), and each partition can be shared by different tenant functions through vGPU time-slicing, thereby reducing resource waste caused by small kernels. Although not perfect, it is the best effort we can make in our scenario.

**Performance Interference**: GPU interference has been widely discussed in prior work [12, 54], including unbalanced SM core utilization across concurrent kernels and contention for memory and PCIe bandwidth. Since vGPU uses a non-preemptive kernel execution model, the performance degradation caused by resource contention is not significant. However, the context switch (hundreds of microseconds) and the queuing delay may cause significant performance degradation for user functions. A large kernel task (>20ms) may take several scheduling periods to complete. The current version of gShare uses a native approach to control the number of launched vGPU instances on a physical GPU with execution-time prediction, which is demonstrated to work in practice. To alleviate this problem, several learning-based performance prediction methods [86] or QoS-aware feedback control on resource allocation [18, 107] can be used to reduce the function performance fluctuations, which is worth further exploration in future work.




(a) The optimus solution (b) Comparison of solving time

**Figure 10.** Analysis of the performance roofline.

## 6 Related Work

We summarize the related work as follows. The key distinctions between gShare and some of them are in Table 4.

**Serverless GPU Functions**: Serverless model serving has been extensively studied in the past few years [5, 26, 32, 34, 48, 67, 71, 80, 84, 105]. However, early approaches are limited to CPU-only functions and suffer from high latency in model serving services [99]. A few studies have recently proposed GPU-enabled FaaS platforms [21, 30, 31, 99], which are usually built on a host-container architecture and do not align with the VM isolation requirements in a multi-tenant FaaS cluster. Although cloud providers are beginning to explore on-demand GPU services [9, 16], their coarse-grained resource provisioning and coupled CPU-GPU management have led to non-trivial operational costs. To the best of our knowledge, gShare is the first work to introduce a low-overhead, fine-grained GPU virtualization solution tailored for a multi-tenant FaaS environment, which can also help cloud providers explore the upper bound of cost-efficient GPU function optimization.

**GPU Sharing & Scheduling**: The work most closely related to gShare is FaaSwap [103], a GPU proxy-based system that employs model swapping to share GPUs across different models. Other systems, such as Clipper [19], Nexus [78], and INFaaS [71], focus on improving resource efficiency through dynamic GPU allocation and request scheduling. However, these frameworks are primarily designed for "serverful" environments and have not been evaluated under serverless workloads. Several studies, including Swayam [32], BATCH [5], INFless [99] and Dilu [52], leverage dynamic batching to improve the throughput of the model serving system. While they rely on keep-alive policies to balance performance and cost, this can result in substantial idle waste under fluctuating workloads. Moreover, many prior works

[12–14, 20, 33, 47, 51, 58, 83, 92, 94] employ techniques such as NVIDIA MPS or MIG to enable temporal-spatial GPU sharing. However, they are constrained by hardware and isolation requirements and cannot be applied to the public cloud.

**Other Techniques**: There are also a number of works that optimize GPU functions. For example, using techniques like startup acceleration [6, 7, 22, 89, 93, 97], function prewarming [55, 77, 84, 101, 109], and sandbox reusing [4, 41, 50, 66] to reduce the startup time of GPU functions, thereby reducing the keep-alive cost. Approaches such as tensor memory sharing [48] and model compression or transformation [34, 96] can help reduce model memory footprints and resource usage. Emerging technologies such as CXL-based disaggregated memory pools [85] can also be used in model swapping and reduce memory costs. Additionally, leveraging CPU-GPU interchangeability and collaborative serving technologies [11, 45, 88, 106] may enhance the model swapping efficiency and function performance in `gShare`. These approaches are orthogonal to `gShare` and can be integrated to increase function density and further improve cloud cost on FaaS platforms.

## 7 Conclusion

This paper presents `gShare`, a server-level GPU function management strategy, which is simple, efficient, and easy to use in practice. `gShare` aims to fill the gap in research on GPU virtualization and sharing for multi-tenant FaaS scenarios and argues for a decoupled CPU-GPU management approach to provide on-demand, cost-efficient GPU resources for model-serving services. In future work, we plan to integrate resource-procurement policies into cloud systems and explore cluster-level server-scaling mechanisms to further reduce datacenter costs and energy consumption in FaaS platforms.

## 8 Acknowledgements

We are grateful to ASPLOS'26 anonymous reviewers and our shepherd, Dong Du, for professional and insightful comments. This work is supported by the the National Natural Science Foundation of China under grant U25B2021.

## References

[1] Jacob Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. 2012. The case for GPGPU spatial multitasking. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*. IEEE Computer Society, 79–90. doi:10.1109/HPCA.2012.6168946

[2] Advanced Micro Devices, Inc. (AMD). 2024. *ROCm Software Stack Documentation*. https://rocm.docs.amd.com Accessed: 2024-06-05.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[5] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 69. doi:10.1109/SC41405.2020.00073

[6] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. 2025. CXLfork: Fast Remote Fork over CXL Fabrics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 210–226. doi:10.1145/3676641.3715988

[7] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS made fast using snapshot-based VMs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 730–746. doi:10.1145/3492321.3524270

[8] Moiz Arif, Kevin Assogba, and M. Mustafa Rafique. 2022. Canary: Fault-Tolerant FaaS for Stateful Time-Sensitive Applications. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*, Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode (Eds.). IEEE, 41:1–41:16. doi:10.1109/SC41404.2022.00046

[9] Microsoft Azure. 2025. Using serverless GPUs in Azure Container Apps. https://learn.microsoft.com/en-us/azure/container-apps/gpu-serverless-overview.

[10] Benjamin Carver, Runzhou Han, Jingyuan Zhang, Mai Zheng, and Yue Cheng. 2023. $\lambda$FS: A Scalable and Elastic Distributed File System Metadata Service using Serverless Functions. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Michael M. Swift, and Natalie D. Enright Jerger (Eds.). ACM, 394–411. doi:10.1145/3623278.3624765

[11] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. 2025. KTransformers: Unleashing the Full Potential of CPU/GPU Hybrid Inference for MoE Models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs (Eds.). ACM, 1014–1029. doi:10.1145/3731569.3764843

[12] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 17–32. doi:10.1145/3037697.3037700

[13] Wenyan Chen, Zizhao Mo, Huanle Xu, Kejiang Ye, and Chengzhong Xu. 2023. Interference-aware Multiplexing for Deep Learning in GPU

Clusters: A Middleware Approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, Dorian Arnold, Rosa M. Badia, and Kathryn M. Mohror (Eds.). ACM, 30:1–30:15. doi:10.1145/3581784.3607060

[14] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[15] Alibaba Cloud. 2025. Alibaba Function Compute (FC). https://www.aliyun.com/product/fc.

[16] Aliyun Cloud. 2025. cGPU overview. https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/cgpu-overview.

[17] Alibaba Cloud. 2025. Function Compute Service Pricing & Purchasing Methods. https://www.alibabacloud.com/product/function-compute/pricing.

[18] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 477–491. doi:10.1145/3419111.3421285

[19] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[20] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 492–506. doi:10.1145/3419111.3421284

[21] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 797–813. doi:10.1145/3503222.3507732

[22] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 467–481. doi:10.1145/3373376.3378512

[23] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010, June 28 - July 2, 2010, Caen, France*, Waleed W. Smari and John P. McIntire (Eds.). IEEE, 224–231. doi:10.1109/HPCS.2010.5547126

[24] Hugging Face. 2025. The AI community building the future. https://huggingface.co/models?sort=downloads.

[25] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[26] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 135–153. https://www.usenix.org/conference/osdi24/presentation/fu

[27] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 386–400. doi:10.1145/3445814.3446757

[28] Google. 2022. gVisor: Application Kernel for Containers. https://gvisor.dev/.

[29] GoogleCloud. 2025. Serverless Optical Character Recognition (OCR) Tutorial. https://cloud.google.com/functions/docs/tutorials/ocr.

[30] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 266–280. doi:10.1145/3575693.3575721

[31] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. 2023. FaST-GShare: Enabling Efficient Spatio-Temporal GPU Sharing in Serverless Computing for Deep Learning Inference. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA, August 7-10, 2023*. ACM, 635–644. doi:10.1145/3605573.3605638

[32] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch (Eds.). ACM, 109–120. doi:10.1145/3135974.3135993

[33] Ziyi Han, Ruiting Zhou, Chengzhong Xu, Yifan Zeng, and Renli Zhang. 2024. InSS: An Intelligent Scheduling Orchestrator for Multi-GPU Inference With Spatio-Temporal Sharing. *IEEE Transactions on Parallel and Distributed Systems: A Publication of the IEEE Computer Society* 10 (2024), 35. doi:10.1109/TPDS.2024.3430063

[34] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 1039–1053. doi:10.1145/3627703.3629567

[35] Docker Inc. 2025. docker container pause. https://docs.docker.com/engine/reference/commandline/container_pause/.

[36] Andy Jassy. 2025. Amazon AWS ReInvent Keynote. https://www.youtube.com/watch?v=ZOIkOnW640A.

[37] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 152–166. doi:10.1145/

3445814.3446701

[38] Kaggle. 2025. Integrating TensorFlow Hub with Kaggle Models. https://www.kaggle.com/models?tfhub-redirect=true&owner-type=organization.

[39] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. 2022. Jiffy: elastic far-memory for stateful serverless analytics. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 697–713. doi:10.1145/3492321.3527539

[40] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[41] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[42] Kubernetes. 2025. Production-Grade Container Orchestration. https://kubernetes.io/.

[43] AWS Lambda. 2025. Keeping Functions Warm. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html.

[44] AWS Lambda. 2025. Serverless Compute - Amazon Web Services. https://aws.amazon.com/cn/lambda/pricing/.

[45] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: compute allocation in hybrid clusters. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 31:1–31:16. doi:10.1145/3342195.3387547

[46] Kevin Lee, Vijay Rao, and William Arnold. 2025. Accelerating Facebook's Infrastructure with Application-Specific Hardware. https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/.

[47] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, Ada Gavrilovska, Deniz Altinbüken, and Carsten Binnig (Eds.). ACM, 173–189. doi:10.1145/3542929.3563510

[48] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association. https://www.usenix.org/conference/atc22/presentation/li-jie

[49] Tao Li, Yongkun Li, Wenzhe Zhu, Yinlong Xu, and John C. S. Lui. 2024. MinFlow: High-performance and Cost-efficient Data Passing for I/O-intensive Stateful Serverless Analytics. In *22nd USENIX Conference on File and Storage Technologies, FAST 2024, Santa Clara, CA, USA, February 27-29, 2024*, Xiaosong Ma and Youjip Won (Eds.). USENIX Association, 311–327. https://www.usenix.org/conference/fast24/presentation/li

[50] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[51] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU Memory Sharing for Concurrent DNN Training. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 161–175. https://www.usenix.org/conference/atc21/presentation/lim

[52] Cunchi Lv, Xiao Shi, Zhengyu Lei, Jinyue Huang, Wenting Tan, Xiaohui Zheng, and Xiaofang Zhao. 2025. Dilu: Enabling GPU Resourcing-on-Demand for Serverless DL Serving via Introspective Elasticity. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 311–325. doi:10.1145/3669940.3707251

[53] LWN.net. 2025. vfio/mdev: IOMMU aware mediated device. https://lwn.net/Articles/783892/. (2025).

[54] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 289–304. https://www.usenix.org/conference/nsdi20/presentation/mahajan

[55] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[56] Philipp Muens. 2025. Serverless Facebook Messenger Bot. https://github.com/pmuens/serverless-facebook-messenger-bot.

[57] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 115–130. doi:10.1145/3318464.3389758

[58] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 595–610. doi:10.1145/3600006.3613163

[59] NVIDIA. 2025. CUDA checkpoint and restore utility. https://github.com/NVIDIA/cuda-checkpoint.

[60] NVIDIA. 2025. Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[61] NVIDIA. 2025. Multi-Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html.

[62] NVIDIA. 2025. NVIDIA Ampere GPU Architecture Tuning Guide. https://docs.nvidia.com/cuda/archive/11.0/ampere-tuning-guide/index.html.

[63] NVIDIA. 2025. Scheduling policy of NVIDIA's vGPU. https://docs.nvidia.com/vgpu/faq/latest/deployment.html.

[64] NVIDIA. 2025. Time-Slicing GPUs in Kubernetes. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html.

[65] NVIDIA. 2025. Using NVIDIA vGPU. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/install-gpu-operator-vgpu.html.

[66] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[67] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. 2023. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*. ACM, 324–340. doi:10.1145/3620678.3624664

[68] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[69] Alessandro Randazzo and Ilenia Tinnirello. 2019. Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In *Sixth International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2019, Granada, Spain, October 22-25, 2019*, Mohammad A. Alsmirat and Yaser Jararweh (Eds.). IEEE, 209–214. doi:10.1109/IOTSMS48152.2019.8939164

[70] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 446–459. doi:10.1109/ISCA45697.2020.00045

[71] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[72] Steven S. Seiden. 2002. On the online bin packing problem. *J. ACM* 49, 5 (2002), 640–671. doi:10.1145/585265.585269

[73] Amazon Web Services. 2025. Amazon EC2 T3 Instances. https://aws.amazon.com/ec2/instance-types/t3/.

[74] Amazon Web Services. 2025. Amazon GPU EC2 Instance Pricing. https://calculator.aws/#/createCalculator/ec2-enhancement.

[75] Amazon Web Services. 2025. AWS Lambda overview. https://aws.amazon.com/lambda/.

[76] Amazon Web Sevice. 2025. Serverless Application Lens: Alexa Skills. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/alexa-skills.html.

[77] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[78] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 322–337. doi:10.1145/3341301.3359658

[79] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2012. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Trans. Computers* 61, 6 (2012), 804–816. doi:10.1109/TC.2011.112

[80] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E. Gonzalez, and Joseph M. Hellerstein. 2020. Optimizing Prediction Serving on Low-Latency Serverless Dataflow. *CoRR* abs/2007.05832 (2020). arXiv:2007.05832 https://arxiv.org/abs/2007.05832

[81] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, Yan Solihin and Mark A. Heinrich (Eds.). ACM, 34:1–34:15. doi:10.1145/3579371.3589069

[82] Radostin Stoyanov, Viktória Spišaková, Jesus Ramos, Steven Gurfinkel, Andrei Vagin, Adrian Reber, Wesley Armour, and Rodrigo Bruno. 2025. CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads. arXiv:2502.16631 [cs.DC] https://arxiv.org/abs/2502.16631

[83] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 1075–1092. doi:10.1145/3627703.3629578

[84] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. 2024. Pre-Warming is Not Enough: Accelerating Serverless Inference With Opportunistic Pre-Loading. In *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC 2024, Redmond, WA, USA, November 20-22, 2024*. ACM, 178–195. doi:10.1145/3698038.3698509

[85] Wenda Tang, Ying Han, Tianxiang Ai, Guanghui Li, Bin Yu, and Xin Yang. 2024. Yggdrasil: Reducing Network I/O Tax with (CXL-Based) Distributed Shared Memory. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 597–606. doi:10.1145/3673038.3673138

[86] Wenda Tang, Yutao Ke, Senbo Fu, Hongliang Jiang, Junjie Wu, Qian Peng, and Feng Gao. 2022. Demeter: QoS-aware CPU scheduling to reduce power consumption of multiple black-box workloads. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 31–46. doi:10.1145/3542929.3563476

[87] Slurm Development Team. 2025. Slurm Quickstart Guide. https://slurm.schedmd.com/quickstart.html.

[88] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[89] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 559–572. doi:10.1145/3445814.3446714

[90] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[91] Bin Wang, Ahmed Ali-Eldin, and Prashant J. Shenoy. 2021. LaSS: Running Latency Sensitive Serverless Computations at the Edge. In *HPDC '21: The 30th International Symposium on High-Performance Parallel and Distributed Computing, Virtual Event, Sweden, June 21-25, 2021*, Erwin Laure, Stefano Markidis, Ana Lucia Verbanescu, and Jay F. Lofstead (Eds.). ACM, 239–251. doi:10.1145/3431379.3460646

[92] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. 2021. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. In *Proceedings of the Fourth Conference on Machine Learning and Systems, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2021/hash/099268c3121d49937a67a052c51f865d-Abstract.html

[93] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[94] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[95] Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2022. Serverless Data Science - Are We There Yet? A Case Study of Model Serving. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1866–1875. doi:10.1145/3514221.3517905

[96] Qizheng Yang, Tianyi Yang, Mingcan Xiang, Lijun Zhang, Haoliang Wang, Marco Serafini, and Hui Guan. 2024. GMorph: Accelerating Multi-DNN Inference via Model Fusion. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 505–523. doi:10.1145/3627703.3650074

[97] Yanning Yang, Dong Du, Haitao Song, and Yubin Xia. 2024. On-demand and Parallel Checkpoint/Restore for GPU Applications. In *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC 2024, Redmond, WA, USA, November 20-22, 2024*. ACM, 415–433. doi:10.1145/3698038.3698510

[98] Yanan Yang, Laiping Zhao, Yiming Li, Shihao Wu, Yuechan Hao, Yuchi Ma, and Keqiu Li. 2023. Flame: A Centralized Cache Controller for Serverless Computing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Michael M. Swift, and Natalie D. Enright Jerger (Eds.). ACM, 153–168. doi:10.1145/3623278.3624769

[99] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 768–781. doi:10.1145/3503222.3507709

[100] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2017. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, Vivek Sarkar and Lawrence Rauchwerger (Eds.). ACM, 221–234. doi:10.1145/3018743.3018754

[101] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 335–350. doi:10.1145/3617232.3624871

[102] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 1489–1504. https://www.usenix.org/conference/nsdi23/presentation/yu

[103] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. 2023. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. *CoRR* abs/2306.03622 (2023). arXiv:2306.03622 doi:10.48550/ARXIV.2306.03622

[104] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, Haoran Yang, and Yu Ding. 2025. Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference. In *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*, Deniz Altinbüken and Ryan Stutsman (Eds.). USENIX Association, 597–612. https://www.usenix.org/conference/atc25/presentation/yu

[105] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[106] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 951–965. https://www.usenix.org/conference/atc18/presentation/zhang-minjia

[107] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: component-distinguishable workload deployment in datacenters. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 19:1–19:17. doi:10.1145/3342195.3387534

[108] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. 2023. BeeHive: Sub-second Elasticity for Web Services with Semi-FaaS Execution. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 74–87. doi:10.1145/3575693.3575752

[109] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2023. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 1–14. doi:10.1145/3567955.3567960

**Table 5.** Experimental testbed configuration.

| | |
|---|---|
| $N$ | Number of GPU pools in the worker node |
| $M$ | Capacity of the vGPU pool |
| $K$ | Number of time slots divided on each GPU slice |
| $R^t$ | Set of requests to be scheduled, with size $\lvert R^t \rvert$ |
| $r_k$ | The $k$-th arriving request |
| $F_{u,v}$ | If $r_u$ and $r_v$ belong to the same tenant function, 1 else 0 |
| $I_k$ | Model swap-in time for the function of request $r_k$ |
| $(K_k, \theta_k, C_k)$ | Arrival time, latency SLO target, and resource requirement of $r_k$ |
| $d_{\text{proc}}(r_k)$ | Execution time of request $r_k$ under resource $C_k$ |
| $d_{\text{init}}(r_k)$ | Initialization time before executing $r_k$ |
| $F_{\delta,\text{sum}}$ | Total GPU allocation in time slot $\delta$ during cycle $t$ |
| $a_{j,k}(\delta)$ | If $r_k$ starts executing on GPU slice $j$ at time slot $\delta$, 0 else 1 |
| $s^j_{u,v}$ | If $r_u$, $r_v$ on the same GPU slice and $r_v$ executing next to $r_u$, 0 else 1 |
| $q$ | Number of activated GPU slices in the worker node during cycle $t$ |

# A Model Statistics

We collect the top 3,000 most-downloaded models from two of the world's largest ML model repositories: TensorFlow Hub (TH) [38] and Hugging Face (HF) [24], which host about 1.94 million models from text, image, video, and natural language processing. In Table 6, we select 100 representative models (by sampling the 3,000 models every 30 steps) and present their characterization, including the model name, parameter size, precision type, memory consumption, and reference. The memory usage bars are scaled logarithmically to reduce the visual difference between the maximum and minimum values, where models with less than 1GB of memory usage are colored orange.

# B Problem Formulation

As mentioned in § 3.1, we first partition the GPU devices on each worker node into multiple physical GPU slices (with different memory sizes such as 128MB, 512MB, 1024MB, etc.). With GPU virtualization techniques, GPU slices can be mapped to user functions as vGPU devices, and multiple tenants can share the same GPU slice with the vGPU reclamation and reallocation mechanism. For each deployed model serving function, the user request arrives in a time sequence with a latency target (e.g., 200ms). Our goal is to schedule requests on a function instance and allocate an available GPU slice for processing, thus minimizing the overall GPU allocation while satisfying the SLO of user latency. Here we present a detailed problem formulation, and the major notations used hereafter are listed in Table 5.

## B.1 Request Scheduling Model

Let $r_k$ denote the $k$-th arriving request, with each $r_k \in R^t$ represented as a 4-tuple $\langle C_k, d_{\text{proc}}(r_k), K_k, \theta_k \rangle$, where $C_k$ is the function vGPU quota corresponding to $r_k$, $d_{\text{proc}}(r_k)$ is the model serving time under $C_k$, $K_k$ is the arrival time, and $\theta_k$ is the latency target. We use a binary variable $a_{j,k}(\delta)$ to indicate whether $r_k$ starts executing at time slot $\delta$ on GPU slice $j$:

$$a_{j,k}(\delta) = \begin{cases} 1, & \text{if } r_k \text{ is scheduled to slot } \delta \text{ on GPU } j \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

We use another binary variable $s^j_{u,v}$ to define the execution order of any two requests $r_u$ and $r_v$. It is 1 if $r_u$ and $r_v$ are scheduled to the same GPU slice and $r_u$ is the nearest prefix execution of $r_v$. Otherwise, $s^j_{u,v}$ is 0. By combining the decision variables $a_{j,k}(\delta)$ and $s^j_{u,v}$, the request execution and vGPU scheduling solution on a worker node can be determined with the following constraints:

First, a request cannot be scheduled before its arrival time:

$$\sum_{j \in M, \delta \in t} a_{j,k}(\delta) \cdot \delta \geq \sum_{j \in M, \delta \in t} a_{j,k}(\delta) \cdot K_k, \quad \forall r_k \in R^t \tag{12}$$

Each request should execute exactly once on a worker node:

$$\sum_{j \in M, \delta \in t} a_{j,k}(\delta) = 1, \quad \forall r_k \in R^t \tag{13}$$

The $s^j_{u,v}$ should satisfy the following constraints:

$$\sum_{r_u, r_v \in R^t} s^j_{u,v} = \sum_{r_k \in R^t, \delta \in t} a_{j,k}(\delta), \quad \forall j \tag{14}$$

$$s^j_{u,v} + s^j_{v,u} = 1, \quad \forall r_u, r_v \in R^t, \forall j \tag{15}$$

$$\sum_{r_u \in R^t} s^j_{u,v} \leq 1, \quad \forall r_v \in R^t, \forall j \tag{16}$$

$$\sum_{r_v \in R^t} s^j_{u,v} \leq 1, \quad \forall r_u \in R^t, \forall j \tag{17}$$

Equations (14-17) ensure a definite execution order for any two adjacent requests $r_u$ and $r_v$ that are on the same GPU slice. Equation (18) ensures the execution has no overlapping for any two requests on each GPU slice:

$$\sum_{\delta \in t} a_{j,v}(\delta) \cdot \delta \geq \sum_{\delta \in t} a_{j,u}(\delta) \cdot \delta + D_u - H \cdot (1 - s^j_{u,v}), \quad \forall r_u, r_v \in R^t, \forall j \tag{18}$$

where $H$ is an extremely large integer.

## B.2 Request Latency Model

The request latency consists of two parts:

(i) Initialization time $d_{\text{init}}(r_k)$: When $r_k$ is scheduled on a GPU slice, if it is the first request processed on this slice or it has a preceding request $r_{k'}$ belongs to a different tenant function, an initialization for model swap-in is required before execution; Otherwise, the initialization phase is skipped. Since the vGPU hot-plugging time is negligible, the initialization time $d_{\text{init}}(r_k)$ can be derived as follows.

$$d_{\text{init}}(r_k) = I_k \cdot (1 - F_{k',k}), \quad \forall r_k \in R^t \tag{19}$$

where $k' = \sum_{r_u \in R^t} s_{u,k} \cdot u$.

(ii) Model serving time $D_k$: The optimal solution is obtained through an offline computation, which means that the request arrival pattern and execution time of all requests are known in advance. We assume that the execution time of the request is independent of the scheduling order.

To satisfy the user latency objective, we have:

$$\left(\sum_{j\in M,\delta\in t} a_{j,k}(\delta)\cdot(\delta-1)\right)\cdot\Delta t+d_{\text{init}}(r_k)+D_k\le\theta_k,\quad \forall r_k\in R^t \tag{20}$$

### B.3 Resource Allocation Model

gShare allows multiple concurrent requests to be executed simultaneously on different vGPU devices (i.e., GPU slices). This requires that the total GPU slice allocation per slot should not exceed the GPU pool capacity. Assume the maximum available resource $M$ of the GPU pool is fixed during cycle $t$. Let $F_{\delta,\text{sum}}=\sum_{j\in M,r_k\in R^t} a_{j,k}(\delta)$ denote the total GPU allocation in time slot $\delta$ during cycle $t$, we have:

$$0\le F_{\delta,\text{sum}}\le M,\quad \forall\delta\in t \tag{21}$$

### B.4 Optimization Target

In the FaaS platform, a GPU slice is considered activated (incurring procurement cost) if it is allocated at any time slot during cycle $t$. In this way, the optimization objective is to minimize the number of allocated GPU slices while satisfying the SLOs for user latency. We use $U_j$ to represent whether the GPU slice $j$ is used during cycle $t$, it is 1 if GPU slice $j$ executes at least one request during cycle $t$ (i.e., $\sum_{\delta\in t,r_k\in R^t} a_{j,k}(\delta)>0$), otherwise $U_j=0$:

$$U_j=\begin{cases}1, & \text{if GPU } j \text{ executes at least one request in cycle } t\\ 0, & \text{otherwise}\end{cases}$$

The objective of the optimization is:

$$\text{Minimize: } \sum_{j\in M} U_j \quad \text{s.t. Eqs. (12-21)}$$

### B.5 Discussion

Note that we use Equations (13) and (20) to simplify the problem solving, which ensures that all requests can be scheduled without SLO violations. However, it is difficult to meet all of the users' latency targets in practice, and the MINLP model will transform into a multi-objective optimization problem where the FaaS platform aims to minimize GPU allocation while satisfying as many users' latency targets as possible. For this goal, we can introduce an activation function $f(x_k)$ to represent the latency target meeting ratio. For example, $f(x_k)$ can be defined as:

$$f(x_k)=\begin{cases}1, & \text{if } x_k>0\\ 0, & \text{otherwise}\end{cases} \tag{22}$$

where $x_k=\theta_k-\left(\sum_{j\in M,\delta\in t} a_{j,k}(\delta)\cdot(\delta-1)\right)\cdot\Delta t-d_{\text{init}}(r_k)-D_k,\quad \forall r_k\in R^t$. A request is marked 1 if it meets the latency target; otherwise, it is 0. Activation functions like $f(x_k)=\frac{1}{1+e^{-x}}, x>0$ are also applicable.

A new objective in Equation (20) can be derived from the activation function, which is:

$$\text{Minimize: } \alpha\cdot\frac{\sum_{x_k\in R^t} f(x_k)}{|R^t|}+(1-\alpha)\cdot\frac{\sum_{j\in M} U_j}{M} \tag{23}$$

where $\alpha\in(0,1)$ is a weighted coefficient to balance the function performance with resource cost, which can be defined according to the business goal of FaaS provider (We do not discuss it here).

### B.6 Solving MINLP with LINGO

We use the LINGO package to solve this formulation on benchmark workloads. Even after incorporating optimizations, such as using a large latency target to ensure the request latency SLOs can be met and assuming users use the same vGPU quota, solving the problem is still expensive. For example, computing the minimum number of vGPUs for 20 requests across 50 time slots takes several minutes, even though the upper bound is only 4 GPUs. The LINGO programming code is listed as follows.

```
/* Definition of problem and variables */
model:
sets:
  Gpu/@file('lingo_data.txt')/:use;
  Req/@file('lingo_data.txt')/:queue,swapTime
      ,arrvTime,procTime,latTarget,startTime;
  Slot/@file('lingo_data.txt')/:gpuAllocSum,
      slotId;
  GRS(Gpu,Req,Slot):sched;
  RR(Req,Req);
  GRR(Gpu,Req,Req):gpuReqBeforeReq;
  GR(Gpu,Req):slotIdVision,slotOcVision;
  GS(Gpu,Slot);
  RS(Req,Slot);
endsets
/* Initialize the input data */
data:
    gpuCap=@file('lingo_data.txt');
    slotCap=@file('lingo_data.txt');
    deltT=@file('lingo_data.txt');
    swapTime=@file('lingo_data.txt');
    arrvTime=@file('lingo_data.txt');
    procTime=@file('lingo_data.txt');
    latTarget=@file('lingo_data.txt');
    slotId=@file('lingo_data.txt');
    queue=@file('lingo_data.txt');
enddata
/* The optimization object and constraints */
min=@sum(Gpu(j):use(j));
@for(Gpu(j):use(j)=@if(@sum(RS(k,s):sched(j,k
    ,s))#gt#0,1,0));
@for(Slot(s):gpuAllocSum(s)=@sum(GR(j,k):
    sched(j,k,s)));
@for(Slot(s):gpuAllocSum(s)<=gpuCap);
@for(GRS(j,k,s):@bin(sched(j,k,s)));
@for(Req(k):@sum(GS(j,s):sched(j,k,s))=1);
@for(GS(j,s):@sum(Req(k):sched(j,k,s))<=1);
@for(Req(k):startTime(k)=@sum(GS(j,s):sched(j
    ,k,s)*slotId(s)));
@for(Req(k):startTime(k)>=arrvTime(k));
@for(GR(j,k):slotIdVision(j,k)=@sum(Slot(s):
    sched(j,k,s)*slotId(s)));
```

**Table 6.** A subset of ML models from TensorFlow Hub and Hugging Face.

| #Id | Model Name | Model Size | Type | Memory Occu. (log) | Ref. | #Id | Model Name | Model Size | Type | Mem Occu. | Ref. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | deepseek-ai/DeepSeek-R1 | 685B | F32 | | HF | 51 | avemio/German-RAG-UAE-LARGE | 335M | I64 | | HF |
| 2 | MiniMaxAI/MiniMax-VL-01 | 456B | F32 | | HF | 52 | tristayqc/my_zh_CN_asr_cv13_model | 319M | F32 | | HF |
| 3 | amd/Llama-3.1-70B-Instruct | 70.6B | F32 | | HF | 53 | ehcalabres/wav2vec2-lg-xlsr-en-speech | 316M | F32 | | HF |
| 4 | 01-ai/Yi-34B-Chat | 34.4B | F32 | | HF | 54 | gigant/romanian-wav2vec2 | 315M | F32 | | HF |
| 5 | leon-se/gemma-3-27b-it-FP8 | 27.4B | F32 | | HF | 55 | jimregan/wav2vec2-large-xlsr-latvian-cv | 315M | I64 | | HF |
| 6 | city96/Qwen-Image-gguf | 20.4B | F32 | | HF | 56 | mrm8488/t5-base-finetuned-question | 297M | F32 | | HF |
| 7 | agentica-org/DeepCoder-14B | 14.8B | F32 | | HF | 57 | visegradmedia-emotion/Emotion | 278M | F32 | | HF |
| 8 | mistral-community/pixtral-12b | 12.7B | I32 | | HF | 58 | TitanML/tiny-mixtral | 247M | F32 | | HF |
| 9 | kosbu/Llama-3.3-70B-Instruct | 11.3B | F32 | | HF | 59 | waveletdeboshir/gigaam-rnnt | 234M | F32 | | HF |
| 10 | BAAI/bge-multilingual-gemma2 | 9.24B | F32 | | HF | 60 | syssec-utd/py312-pylingual-v1-statement | 223M | I64 | | HF |
| 11 | CohereLabs/aya-vision-8b | 8.63B | F32 | | HF | 61 | rinna/japanese-cloob-vit-b-16 | 197M | F32 | | HF |
| 12 | reducto/RolmOCR | 8.29B | F32 | | HF | 62 | Angelakeke/RaTE-NER-Deberta | 184M | F32 | | HF |
| 13 | MLP-KTLim/llama-3-Korean | 8.03B | F32 | | HF | 63 | nnlm-es-dim50-with-normalization | 177M | F32 | | TH |
| 14 | KISTI-KONI/KONI-Llama3.1-8B | 8.03B | F32 | | HF | 64 | resnet_v2_101 | 170M | F32 | | TH |
| 15 | liuhaotian/llava-v1.6-mistral-7b | 7.57B | F32 | | HF | 65 | JackFram/llama-160m | 162M | F32 | | HF |
| 16 | intfloat/e5-mistral-7b-instruct | 7.11B | F32 | | HF | 66 | CIDAS/clipseg-rd64-refined | 151M | F32 | | HF |
| 17 | LongSafari/evo-1-8k-transposon | 6.45B | F32 | | HF | 67 | KoalaAI/Text-Moderation | 139M | F32 | | HF |
| 18 | h2oai/h2o-danube3-4b-chat | 3.96B | I32 | | HF | 68 | lxyuan/distilbert-base-multilingual | 135M | F32 | | HF |
| 19 | ibm-research/PowerMoE-3b | 3.37B | F32 | | HF | 69 | dmargutierrez/distilbert-base | 135M | F32 | | HF |
| 20 | Efficient-Large-Model/gemma | 2.61B | F32 | | HF | 70 | cord-19/swivel-128d | 125M | F32 | | TH |
| 21 | vikhyatk/moondream2 | 1.93B | F32 | | HF | 71 | agufsamudra/indo-sentiment-analysis | 124M | F32 | | HF |
| 22 | utter-project/EuroLLM-1.7B | 1.66B | F32 | | HF | 72 | jinmang2/kpfbert | 113M | F32 | | HF |
| 23 | michaelfeil/mxbai-rerank | 1.54B | F32 | | HF | 73 | utrobinmv/t5_translate_en | 111M | F32 | | HF |
| 24 | infly/inf-retriever-v1-1.5b | 1.54B | F32 | | HF | 74 | hatmimoha/arabic-ner | 110M | F32 | | HF |
| 25 | sentence-transformers/sentence | 1.24B | F32 | | HF | 75 | adeelhasan/email-classifiermodelv1 | 109M | F32 | | HF |
| 26 | bert_en_wwm_uncased_L-24 | 1.2B | F32 | | TH | 76 | HuggingFaceFW/fineweb-edu-classifier | 109M | F32 | | HF |
| 27 | distributed/llama-1b | 1.1B | F32 | | HF | 77 | semi-exemplar-10 | 97M | F32 | | TH |
| 28 | nm-testing/tinyllama-oneshot | 1.1B | F32 | | HF | 78 | Cnam-LMSSC/wav2vec2-french-phonemizer | 94M | F32 | | HF |
| 29 | 5CD-AI/Vintern-1B-v3_5 | 938M | F32 | | HF | 79 | sup-100 | 90M | F32 | | TH |
| 30 | MuRIL | 842M | F32 | | TH | 80 | skshmjn/Pokemon-classifier-gen9-1025 | 86M | F32 | | HF |
| 31 | kotoba-tech/kotoba-whisper-v2.2 | 756M | F32 | | HF | 81 | rizvandwiki/gender-classification | 85M | F32 | | HF |
| 32 | kaitchup/Phi-3-mini-4k-instruct | 684M | F32 | | HF | 82 | rotation | 83M | F32 | | TH |
| 33 | pipecat-ai/smart-turn | 581M | F32 | | HF | 83 | movinet/a5/stream/kinetics-600/classification | 74M | F32 | | TH |
| 34 | nlpai-lab/KURE-v1 | 568M | F32 | | HF | 84 | madhurjindal/autonlp-Gibberish-Detector | 67M | F32 | | HF |
| 35 | heydariAI/persian-eMeddings | 560M | F32 | | HF | 85 | tals/albert-xlarge-vitaminc-mnli | 58M | F32 | | HF |
| 36 | universal-sentence-encoder-qa | 528M | F32 | | TH | 86 | bookbot/distil-ast-audioset | 44M | F32 | | HF |
| 37 | timm/maxvit_xlarge_tf_512 | 476M | F32 | | HF | 87 | inception_v4_quant | 40M | F32 | | TH |
| 38 | nnlm-es-dim128 | 446M | F32 | | TH | 88 | PeiqingYang/MatAnyone | 35M | F32 | | HF |
| 39 | cross-encoder/nli-deberta-v3 | 435M | F32 | | HF | 89 | Isotr0py/test-gguf-sample | 28M | F32 | | HF |
| 40 | bmd1905/vietnamese-correction | 420M | F32 | | HF | 90 | east-text-detector | 23M | F32 | | TH |
| 41 | wiki40b-lm-vi | 415M | F32 | | TH | 91 | vgg19-block4-conv2-unpooling-decoder | 20M | F32 | | TH |
| 42 | wiki40b-lm-it | 414M | F32 | | TH | 92 | mnasnet_1.0_96 | 16M | F32 | | TH |
| 43 | wiki40b-lm-cs | 414M | F32 | | TH | 93 | ganeval-cifar10-convnet | 13M | F32 | | TH |
| 44 | wiki40b-lm-id | 414M | F32 | | TH | 94 | inception_v2_quant | 10M | F32 | | TH |
| 45 | wiki40b-lm-en | 414M | F32 | | TH | 95 | trl-internal-testing/tiny-Qwen2_5 | 9M | F32 | | HF |
| 46 | ByteDance/Dolphin | 398M | F32 | | HF | 96 | handdetector | 6M | F32 | | TH |
| 47 | bert_en_cased_L-12_H-768_A-12 | 386M | F32 | | TH | 97 | spiral/default-fluid-gansn-celebahq64 | 5M | F32 | | TH |
| 48 | biggan-deep-128 | 366M | F32 | | TH | 98 | mobilenet_v1_1.0_224_quantized | 4M | F32 | | TH |
| 49 | garak-llm/roberta-large-snli | 356M | F32 | | HF | 99 | mobilenet_v2_1.0_224_quantized | 3M | F32 | | TH |
| 50 | nasnet/large | 338M | F32 | | TH | 100 | iris | 2M | F32 | | TH |

```
@for(GR(j,k):slotOcVision(j,k)=@sum(Slot(s):
    sched(j,k,s)));
@for(GRR(j,u,v):gpuReqBeforeReq(j,u,v)=@if(
    slotIdVision(j,u)#lt#slotIdVision(j,v),
    slotOcVision(j,u)*slotOcVision(j,v)
    ,99999));
@for(GRR(j,u,v):slotIdVision(j,v)>=
    slotIdVision(j,u)+swapTime(u)+procTime(u)
    +(1-gpuReqBeforeReq(j,u,v)));
@for(Req(k):startTime(k)+swapTime(k)+procTime
    (k)<=arrvTime(k)+latTarget(k));
@for(Req(k):startTime(k)<=slotCap);
end
 /* The end */
```

**Listing 1.** The LINGO code for solving this MINLP model.

```
1 2 3 ~
1 2 3 4 5 6 7 8 9 ~
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
    19 20 ~
3 ~
20 ~
1 ~
1 1 1 1 1 1 1 1 1 ~
2 3 6 9 10 13 15 17 18 ~
1 2 1 1 1 1 1 1 1 ~
2 3 4 2 3 4 2 4 3 ~
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
    19 20 ~
1 2 3 4 5 6 7 8 9 ~
```

**Listing 2.** An example of lingo_data.txt in Figure 1(d).